你真的理解 setState 吗?

在 React 日常的使用中，一个很重要的点就是，不要直接去修改 state。例如：this.state.count = 1 是无法触发 React 去更新视图的。因为 React 的机制规定，一个 state 的更新，首先需要调用 setState 方法。

```
this.setState({
    count: 1
})
```

这样便能触发重新渲染。稍有经验的开发者会知道，setState 方法其实是“异步” 的。即立马执行之后，是无法直接获取到最新的 state 的，需要经过 React 对 state 的所有改变进行合并处理之后，才会去计算新的虚拟 dom，再根据最新的虚拟 dom 去重新渲染真实 dom。

```
class App extends Component {
        state = {
        count: 0
        }

    componentDidMount(){
        this.setState({count: this.state.count + 1})
        console.log(this.state.count) // 0
    }

    render(){
        ...
    }
}
```

setState 真的是异步的吗？

这两天自己简单的看了下 setState 的部分实现代码，在这边给到大家一个自己个人的见解，可能文字或图片较多，没耐心的同学可以直接跳过看总结(**源码版本是 16.4.1**)。

看之前，为了方便理解和简化流程，我们默认 react 内部代码执行到 performWork 、performWorkOnRoot、performSyncWork、performAsyncWork 这四个方法的时候，就是 react 去 update 更新并且作用到 UI 上。

一、合成事件中的 setState

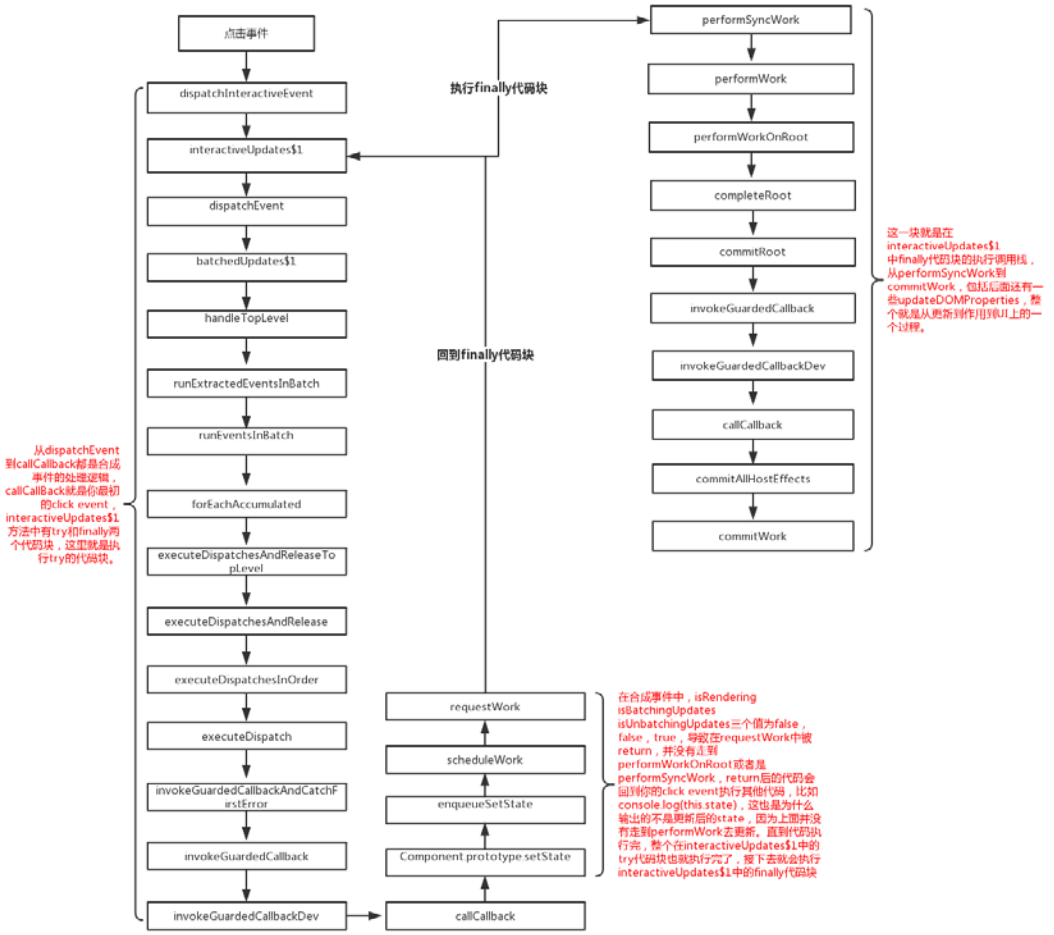
首先得了解一下什么是合成事件，react 为了解决跨平台，兼容性问题，自己封装了一套事件机制，代理了原生的事件，像在 jsx 中常见的 onClick、onChange 这些都是合成事件。

```
class App extends Component {

  state = { val: 0 }

  increment = () => {
    this.setState({ val: this.state.val + 1 })
    console.log(this.state.val) // 输出的是更新前的 val --> 0
  }
  render() {
    return (
      <div onClick={this.increment}>
        {`Counter is: ${this.state.val}`}
      </div>
    )
  }
}
```

合成事件中的 setState 写法比较常见，点击事件里去改变 this.state.val 的状态值，在 increment 事件中打个断点可以看到调用栈，这里我贴一张自己画的流程图：



从 dispatchInteractiveEvent 到 callCallBack 为止，都是对合成事件的处理和执行，从 setState 到 requestWork 是调用 this.setState 的逻辑，这边主要看下 requestWork 这个函数（从 dispatchEvent 到 requestWork 的调用栈是属于 interactiveUpdates$1 的 try 代码块，下文会提到）。

```
function requestWork(root, expirationTime) {
  addRootToSchedule(root, expirationTime);

  if (isRendering) {
    // Prevent reentrancy. Remaining work will be scheduled at the
end of
    // the currently rendering batch.
    return;
  }

  if (isBatchingUpdates) {
    // Flush work at the end of the batch.
    if (isUnbatchingUpdates) {
```

```
      // ...unless we're inside unbatchedUpdates, in which case we
should
      // flush it now.
      nextFlushedRoot = root;
      nextFlushedExpirationTime = Sync;
      performWorkOnRoot(root, Sync, false);
    }
    return;
  }

  // TODO: Get rid of Sync and use current time?
  if (expirationTime === Sync) {
    performSyncWork();
  } else {
    scheduleCallbackWithExpiration(expirationTime);
  }
}
```

在 requestWork 中有三个 if 分支，三个分支中有两个方法 performWorkOnRoot 和 performSyncWork ，就是我们默认的 update 函数，但是在合成事件中，走的是第二个 if 分支，第二个分支中有两个标识 isBatchingUpdates 和 isUnbatchingUpdates 两个初始值都为 false ，但是在 interactiveUpdates$1 中会把 isBatchingUpdates 设为 true ，下面就是 interactiveUpdates$1 的代码：

```
function interactiveUpdates$1(fn, a, b) {
  if (isBatchingInteractiveUpdates) {
    return fn(a, b);
  }
  // If there are any pending interactive updates, synchronously
flush them.
  // This needs to happen before we read any handlers, because the
effect of
  // the previous event may influence which handlers are called
during
  // this event.
  if (!isBatchingUpdates && !isRendering &&
lowestPendingInteractiveExpirationTime !== NoWork) {
    // Synchronously flush pending interactive updates.
    performWork(lowestPendingInteractiveExpirationTime, false, null);
    lowestPendingInteractiveExpirationTime = NoWork;
  }
```

```
  var previousIsBatchingInteractiveUpdates =
isBatchingInteractiveUpdates;
  var previousIsBatchingUpdates = isBatchingUpdates;
  isBatchingInteractiveUpdates = true;
  isBatchingUpdates = true;  // 把 requestWork 中的 isBatchingUpdates
标识改为 true
  try {
    return fn(a, b);
  } finally {
    isBatchingInteractiveUpdates =
previousIsBatchingInteractiveUpdates;
    isBatchingUpdates = previousIsBatchingUpdates;
    if (!isBatchingUpdates && !isRendering) {
      performSyncWork();
    }
  }
}
```

在这个方法中把 isBatchingUpdates 设为了 true ,导致在 requestWork 方法中， isBatchingUpdates 为 true ，但是 isUnbatchingUpdates 是 false ，而被直接 return 了。

那 return 完的逻辑回到哪里呢，最终正是回到了 interactiveUpdates 这个方法，仔细看一眼，这个方法里面有个 try finally 语法，前端同学这个其实是用的比较少的，简单的说就是会先执行 try 代码块中的语句，然后再执行 finally 中的代码，而 fn(a, b) 是在 try 代码块中，刚才说到在 requestWork 中被 return 掉的也就是这个 fn（上文提到的 从 dispatchEvent 到 requestWork 的一整个调用栈）。

所以当你在 increment 中调用 setState 之后去 console.log 的时候，是属于 try 代码块中的执行，但是由于是合成事件，try 代码块执行完 state 并没有更新，所以你输入的结果是更新前的 state 值，这就导致了所谓的"异步"，但是当你的 try 代码块执行完的时候（也就是你的 increment 合成事件），这个时候会去执行 finally 里的代码，在 finally 中执行了 performSyncWork 方法，这个时候才会去更新你的 state 并且渲染到 UI 上。

二、生命周期函数中的 setState

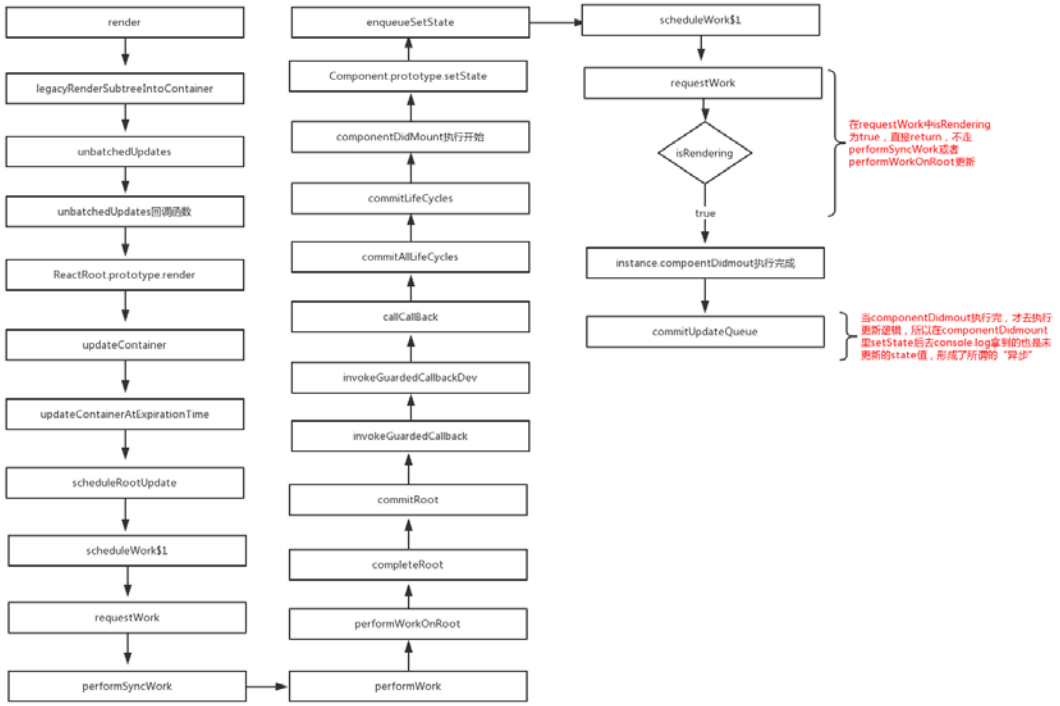
```
class App extends Component {

  state = { val: 0 }

 componentDidMount() {
```

```
    this.setState({ val: this.state.val + 1 })
    console.log(this.state.val) // 输出的还是更新前的值 --> 0
  }
  render() {
    return (
      <div>
        {`Counter is: ${this.state.val}`}
      </div>
    )
  }
}
```

钩子函数中 setState 的调用栈：



其实还是和合成事件一样，当 componentDidmount 执行的时候，react 内部并没有更新，执行完 componentDidmount 后才去 commitUpdateQueue 更新。这就导致你在 componentDidmount 中 setState 完去 console.log 拿的结果还是更新前的值。

三、原生事件中的 setState

```
class App extends Component {
```

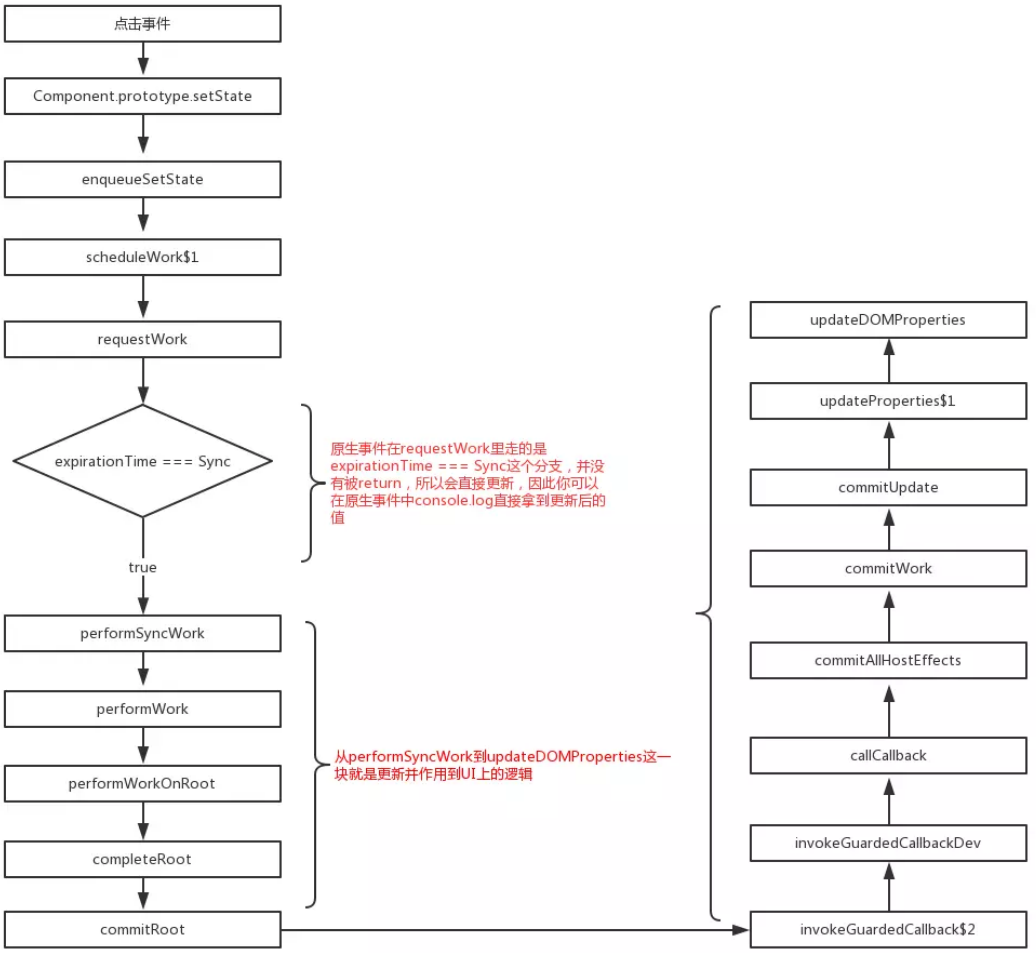
```
  state = { val: 0 }

  changeValue = () => {
    this.setState({ val: this.state.val + 1 })
    console.log(this.state.val) // 输出的是更新后的值 --> 1
  }

 componentDidMount() {
    document.body.addEventListener('click', this.changeValue, false)
 }

  render() {
    return (
      <div>
        {`Counter is: ${this.state.val}`}
      </div>
    )
  }
}
```

原生事件是指非 react 合成事件，原生自带的事件监听 addEventListener ，或者也可以用原生 js、jq 直接 document.querySelector().onclick 这种绑定事件的形式都属于原生事件。



原生事件的调用栈就比较简单了，因为没有走合成事件的那一大堆，直接触发 click 事件，到 requestWork ，在 requestWork 里由于 expirationTime === Sync 的原因，直接走了 performSyncWork 去更新，并不像合成事件或钩子函数中被 return，所以当你在原生事件中 setState 后，能同步拿到更新后的 state 值。

四、setTimeout 中的 setState

```
class App extends Component {

  state = { val: 0 }

 componentDidMount() {
    setTimeout(_ => {
      this.setState({ val: this.state.val + 1 })
      console.log(this.state.val) // 输出更新后的值 --> 1
    }, 0)
```

```
  }

  render() {
    return (
      <div>
        {`Counter is: ${this.state.val}`}
      </div>
    )
  }
}
```

在 setTimeout 中去 setState 并不算是一个单独的场景，它是随着你外层去决定的，因为你可以在合成事件中 setTimeout ，可以在钩子函数中 setTimeout ，也可以在原生事件 setTimeout，但是不管是哪个场景下，基于 event loop 的模型下， setTimeout 中里去 setState 总能拿到最新的 state 值。

举个栗子，比如之前的合成事件，由于你是 setTimeout(_ => { this.setState()}, 0) 是在 try 代码块中,当你 try 代码块执行到 setTimeout 的时候，把它丢到列队里，并没有去执行，而是先执行的 finally 代码块，等 finally 执行完了， isBatchingUpdates 又变为了 false ，导致最后去执行队列里的 setState 时候， requestWork 走的是和原生事件一样的 expirationTime === Sync if 分支，所以表现就会和原生事件一样，可以同步拿到最新的 state 值。

五、setState 中的批量更新

```
class App extends Component {

  state = { val: 0 }

  batchUpdates = () => {
    this.setState({ val: this.state.val + 1 })
    this.setState({ val: this.state.val + 1 })
    this.setState({ val: this.state.val + 1 })
  }

  render() {
    return (
      <div onClick={this.batchUpdates}>
        {`Counter is ${this.state.val}`} // 1
      </div>
    )
```

```
  }
}
```

上面的结果最终是 1，在 setState 的时候 react 内部会创建一个 updateQueue ，通过 firstUpdate 、 lastUpdate 、 lastUpdate.next 去维护一个更新的队列，在最终的 performWork 中，相同的 key 会被覆盖，只会对最后一次的 setState 进行更新，下面是部分实现代码：

```
function createUpdateQueue(baseState) {
  var queue = {
    expirationTime: NoWork,
    baseState: baseState,
    firstUpdate: null,
    lastUpdate: null,
    firstCapturedUpdate: null,
    lastCapturedUpdate: null,
    firstEffect: null,
    lastEffect: null,
    firstCapturedEffect: null,
    lastCapturedEffect: null
  };
  return queue;
}

function appendUpdateToQueue(queue, update, expirationTime) {
  // Append the update to the end of the list.
  if (queue.lastUpdate === null) {
    // Queue is empty
    queue.firstUpdate = queue.lastUpdate = update;
  } else {
    queue.lastUpdate.next = update;
    queue.lastUpdate = update;
  }
  if (queue.expirationTime === NoWork || queue.expirationTime >
expirationTime) {
    // The incoming update has the earliest expiration of any update
in the
    // queue. Update the queue's expiration time.
    queue.expirationTime = expirationTime;
  }
}
```

看个 

```
class App extends React.Component {
  state = { val: 0 }

  componentDidMount() {
    this.setState({ val: this.state.val + 1 })
    console.log(this.state.val)

    this.setState({ val: this.state.val + 1 })
    console.log(this.state.val)

    setTimeout(_ => {
      this.setState({ val: this.state.val + 1 })
      console.log(this.state.val);

      this.setState({ val: this.state.val + 1 })
      console.log(this.state.val)
    }, 0)
  }

  render() {
    return <div>{this.state.val}</div>
  }
}
```

结合上面分析的，钩子函数中的 setState 无法立马拿到更新后的值，所以前两次都是输出 0，当执行到 setTimeout 里的时候，前面两个 state 的值已经被更新，由于 setState 批量更新的策略， this.state.val 只对最后一次的生效，为 1，而在 setTimmout 中 setState 是可以同步拿到更新结果，所以 setTimeout 中的两次输出 2，3，最终结果就为 0, 0, 2, 3 。

总结 ：

1. setState 只在合成事件和钩子函数中是“异步”的，在原生事件和 setTimeout 中都是同步的。
2. setState 的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的 callback 拿到更新后的结果。

3. setState 的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和 setTimeout 中不会批量更新，在“异步”中如果对同一个值进行多次 setState ， setState 的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时 setState 多个不同的值，在更新时会对其进行合并批量更新。